A Formally Verified SMT Approach to True Concurrency*

Juliana K. F. Bowles[0000−0002−5918−9114] and Marco B. Caminati[0000−0002−4529−5442]

School of Computer Science, University of St Andrews
St Andrews KY16 9SX, United Kingdom
{jkfb|mbc8}@st-andrews.ac.uk

Abstract. Many problems related to distributed and parallel systems, such as scheduling and optimisation, are computationally hard, thereby justifying the adoption of SMT solvers. The latter provide standard arithmetic as interpreted functions, naturally leading to express concurrent executions as a linearly-ordered sequentialisation (or interleaving) of events, which have an obvious correspondence with integer segments and therefore permit to take advantage of such arithmetical capabilities. However, there are alternative semantic approaches (also known as true concurrent) not imposing the extra step of interleaving events, which brings the question of how to computationally exploit SMT solvers in these approaches. This paper presents a solution to this problem, and introduces a metric, made possible by adopting a true concurrent paradigm, which relates mutually distinct solutions of a family of distributed optimisation problems. We also contribute an original, computational definition of degree of parallelism, which we compare with the existing ones. Finally, we use theorem proving to formally certify a basic correctness property of our true concurrent approach.

1 Introduction and Related Work

There are many possible models to capture the behaviour of distributed and parallel systems. Here we use labelled (prime) event structures [24], or *event structures* for short. Event structures have been widely studied in the literature, and have been used to give a true concurrent semantics to process calculi such as CCS, CSP, SCCS and ACP (e.g., [23]). The advantages of prime event structures include their underlying simplicity and how they naturally describe fundamental notions present in behavioural models including sequential, parallel and iterative behaviour (or the unfoldings thereof) as well as nondeterminism (cf. [14]), and are hence our model of choice. Event structures consist of sets of events and

* This research is supported by MRC grant MR/S003819/1 and Health Data Research UK, an initiative funded by UK Research and Innovation, Department of Health and Social Care (England) and the devolved administrations, and leading medical research charities.

binary relations over events to represent these notions. When used to capture the behaviour of a system, for example, event structures can be further equipped with labelling (total or partial) functions that associate information to these events, such as the action performed, action synchronisations, or any additional metric that may be relevant within the context. For instance, in a medical context an event may be associated with taking a particular medication, and a measure of interest may be the efficacy of the chosen medication for the underlying treatment.

Event structures have well-defined composition operators (cf. e.g., [18]). However, these composition mechanisms ignore labels and are hence inadequate for our use here. In practice we are not interested in computing the composition of models, but rather in finding optimal paths of execution in such a composition with respect to a certain measure of interest (usually a certain integer value which we want to maximise or minimise). Again, within a medical context this may be to maximise the efficacy of the medications given to patients following multiple treatment plans (which may be the case if patients have *multimorbidities*, i.e., multiple ongoing chronic conditions). In this case, we are searching for optimal treatment plans which also avoid adverse drug reactions between chosen medications (i.e., maximise efficacy whilst minimising adverse reactions).

When measures of interest can be quantified, we can make use of *SMT (Satisfiability Modulo Theories) solvers* to search for the optimal solution. We have used the SMT solver Z3 [17] in our earlier work (see for instance [5,6,7]). However, in that work we have ignored the true concurrent nature of event structures and treated the inherent parallelism within executions as interleaving.

In this paper, we turn our attention to the problem of computing and selecting trace executions in a *true-concurrent* manner, as opposed to an implicit non-deterministic choice among allowed sequentialisations (or interleaving) of events. While work exists providing theoretical frameworks to characterise and reason about true-concurrency [1], [10], [15], true-concurrent approaches to the computational problem introduced above are rare. Other existing works we are aware of (see, for example, [12]) deal with notions of distance between different event structures, rather than between trace executions of one event structure. In [16] a problem similar to the one considered here is studied, but within the theoretical framework of Petri nets, and without including a numerical interaction between sets of concurrent events to influence the execution choice, as we do here: there, the focus is more on the theoretical study of the complexity associated with checking the optimality of a number of resources (concurrency threshold) for the execution of a given parallel process. Our work presents similar differences with [9], where an encoding of timed BPMN models into rewriting logic is presented; this permits to use the rewriting logic implementation Maude [8] in order to reflect about a notion of parallelism degree.

The contributions we provide are: an SMT-viable model for a true-concurrent handling of trace executions, a notion of metric to be able to choose among such executions, and, at the same time, to quantify the degree of concurrency of a trace execution; finally, a formal verification of a basic sanity property of our approach through the theorem prover Isabelle/HOL, where Isabelle is foundation-agnostic

proof assistant and Isabelle/HOL is the strain based on *higher-order logic* (HOL) [19][1]

This paper is structured as follows. Section 2 describes the formal model used, how configurations and paths/traces of executions are defined, as well as what resources we care to maximise/minimise when searching for the optimal path. Our SMT-based formulation of the problem is shown in Section 3 and how verification is done is described in Section 4. This is followed by concluding remarks in Section 5.

2 Formal Model

In an event structure, we have a set of event occurrences together with binary relations for expressing causal dependency (called *causality*) and nondeterminism (called *conflict*). The causality relation implies a (partial) order among event occurrences, while the conflict relation expresses how the occurrence of certain events excludes the occurrence of others. From the two relations defined over the set of events, a further relation is derived, namely the *concurrency* relation *co*. Two events are concurrent if and only if they are completely unrelated, i.e., neither related by causality nor by conflict.

The formal definition of an event structure, as in [14], is as follows.

Definition 1. *An* event structure *is a triple* $E = (Ev, \rightarrow^*, \#)$ *where* Ev *is a set of events and* $\rightarrow^*, \# \subseteq Ev \times Ev$ *are binary relations called* causality *and* conflict*, respectively. Causality* $\rightarrow^*$ *is a partial order. Conflict* $\#$ *is symmetric and irreflexive, and propagates over causality, i.e.,* $e\#e' \wedge e' \rightarrow^* e'' \Rightarrow e\#e''$ *for all* $e, e', e'' \in Ev$*. Two events* $e, e' \in Ev$ *are* concurrent*,* e *co* e' *iff* $\neg(e \rightarrow^* e' \vee e' \rightarrow^* e \vee e\#e')$*.* $C \subseteq Ev$ *is a* configuration *iff (1)* C *is* conflict-free*:* $\forall e, e' \in C \neg(e\#e')$ *and (2)* downward-closed*:* $e \in C$ *and* $e' \rightarrow^* e$ *implies* $e' \in C$*.*

We assume *discrete* event structures. Discreteness imposes a finiteness constraint on the model, i.e., there are always only a finite number of causally related predecessors to an event, known as the *local configuration* of the event (written $\downarrow e$). A further motivation for this constraint is given by the fact that every execution has a starting point or configuration. A maximal configuration is called a *trace*. An event e may have an immediate successor e' according to the order $\rightarrow^*$, written $e \rightarrow e'$, where $\rightarrow$ denotes *immediate causality* iff there is no possible intermediate event. Fig. 1 shows a visual depiction of a simple event structure.

Here, event e_0 marks the initial event, and events $e_1\#e_8$ are in conflict. According to conflict propagation, events $e_1\#e_9$ are also in conflict. Immediate causality is shown, and any events not related by causality or conflict are concurrent. For instance, e_2 *co* e_6 are concurrent. Two configurations are shown surrounding some of the events: in red to the left (X_1), and in green to the right (X_2). X_1 contains concurrency, whereas X_2 does not. Moreover, $X_1 = \downarrow e_7$ and $X_2 = \downarrow e_{10}$.

[1] We will often write just Isabelle in lieu of Isabelle/HOL.

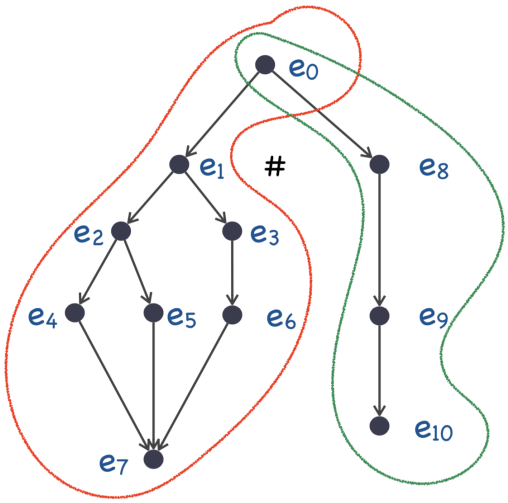

Fig. 1. Example event structure.

To make a connection between the semantic model (an event structure) and the syntactic model (ignored here) it is describing, we need to associate additional information to individual events. Let L be a given set of labels.

Definition 2. *A* labelled event structure *over L is a triple $M = (Ev, \mu, \nu)$ where μ and ν are partial labelling functions $\mu : Ev \to 2^L$ and $\nu : Ev \to \mathbb{N} \times \mathbb{N}$.*

Labelled event structures are event structures enriched with two labelling functions μ and ν. The function μ maps events onto a subset of elements of L. The labels in the set L either denote formulas (constraints over integer variables, e.g., $x > 9$ or $y = 5$), logical propositions (e.g., `pro1`) or actions (e.g., $ma1$). If for an event $e \in Ev$, $\mu(e)$ contains an action $\alpha \in L$, then e denotes an occurrence of that action α. If $\mu(e)$ contains a formula or logical proposition $\varphi \in L$, then φ must hold when e occurs.

The labelling function ν associates to each event its *priority* and *duration*, for instance, $\nu(e) = (p, d)$ indicates that p is the priority and d is the duration associated with e. The higher the value of p, the higher the priority associated to the event. The duration d indicates the time units spent at event e. Sometimes, we will write ν_1 or ν_2 for the function returning only the first or second component of the pair returned by ν.

Giving different priority values to events is meaningful in the presence of alternatives (conflicting events), where the highest value can be used to determine the ideal configuration in a model. For instance, event e_2 may have a higher priority than e_8. Further labels may be added to the framework as partial functions if required. We call a labelled event structure a model in what follows.

In addition, we define a map Γ specifying the level of conflict between event labels as follows.

Definition 3. *Label conflicts are given by a (possibly partial) function* $\Gamma : 2^L \to \mathbb{Z}$.

When we are interested to the value of Γ in a restricted subset of L, we use the notation (l_1, l_2, v), indicating that l_1 and l_2 are in conflict with an *interaction score* of value v. We consider that the lower the value of v the higher the severity of the label conflict. This permits to add further expressiveness to our model: for example, two events may not be in conflict, but the possible choice for their labels could have a bad interaction score, which could lead to alternative solutions. For the event structure in Fig. 1, we assume that $\mu(e_4) = ma_2$ and $\mu(e_3) = mb_2$, and a label conflict such that $\Gamma(\{ma_2, mb_2\}) = -100$.

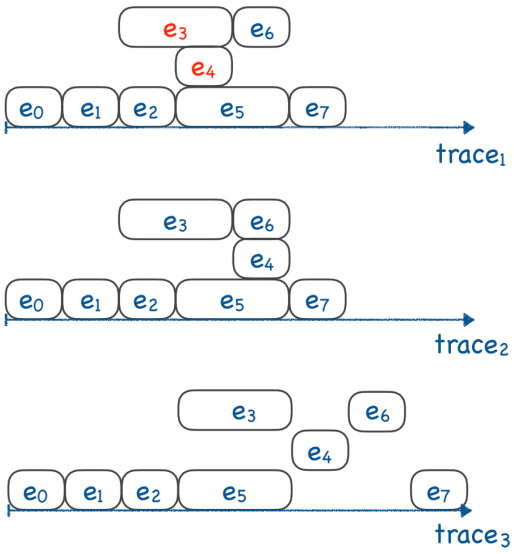

Fig. 2. Possible executions of X_1 with and without label conflicts.

The labels of some of the events (e_3 and e_4 in the example) are conflicting according to Γ. When obtaining the optimal trace execution within the event structure above we need to make sure label inconsistencies are detected and avoided. A trace execution that avoids a label conflict is given by an execution of the configuration X_2. However, this is too restrictive since the priority of e_2 is higher than e_8, and X_1 may hence lead to an overall better trace execution. The label conflicts are only a problem if the events occur simultaneously.

In our earlier approach [7], events in a configuration would be linearised and hence the label conflict automatically avoided. The loss of parallelism would,

however, be too significant, and instead we want to choose trace executions for configurations that avoid label conflicts but achieve the best possible measure of concurrency. Consider a few options for traces execution for X_1 shown in Fig. 2. We show a timeline from left to right and event durations, where overlapping events denote simultaneous execution. As highlighted, $trace_1$ (shorthand name for execution number 1 of the trace) is problematic since the simultaneous execution of e_3 and e_4 would contribute to a label conflict. Both $trace_2$ and $trace_3$ avoid label conflicts, but $trace_2$ has a higher degree of concurrency as it takes less time to execute (more events are executed in parallel). Because the notion of how many events are executed in parallel seems more complicated to define formally, we take the execution time as a definition of degree of parallelism. While the two notions are intuitively strongly correlated, this does not mean that this measure will pick an execution having the highest number of parallel events at all instants.

Let us focus on a given subset of events C (which could be, for example, the events in a trace), and introduce the idea of how to associate to different executions of C a number measuring how parallel they are. The lowest degree of parallelism will correspond to the number $\Sigma_{e \in C} \nu_2(e) + \tau$, given by the sum of all event durations (yielded by the function ν_2) and an additional parameter τ representing idle time in between events. τ is a parameter to describe optional slack in the execution, and can be set to zero when not needed. The highest possible (loose) upper bound to the degree of parallelism will correspond to the maximal duration of an event in C.

In our example, let $C_1 = \{e_2, e_3\}$ and $C_2 = \{e_4, e_5, e_6\}$. The degree of concurrency for different executions of C_1 would be in the range $[2, 3 + \tau]$, where $\nu_2(e_2) = 1$ and $\nu_2(e_3) = 2$; and for C_2 in $[2, 4 + \tau]$ where $\nu_2(e_4) = \nu_2(e_6) = 1$ and $\nu_2(e_5) = 2$. In Fig. 2, executions $trace_1$ and $trace_2$ have the highest degree of concurrency. In the sequel, we will give a precise definition to this idea of degree of concurrency, allowing us to compute its exact value for any execution, picked from the range introduced above.

3 An SMT-oriented Formulation

The efficiency of SMT solvers comes at a price: first, one has to find a formulation of their problem which must be in first-order logic, because SMT-LIB (the standardised input format for SMT solvers [3]) is a first-order language, although with some added interpreted relations (whence the letters M and T in SMT). Secondly, the SMT-LIB code one obtains after managing to do that is typically very little readable for a human, as we will see in this section. We will deal with the first problem in this section, and with the second in Section 4.

We will use the natural index i to range over $(1, \ldots, n)$ where n is the number of models (event structures) we are considering for composition. Since we will need to impose conditions on all of them, in the sequel we will often implicitly quantify over i. We denote by G_i the set of ordered pairs representing the immediate causality relation $\rightarrow_i$; that is, given events e_0, e_1 in the i-th event structure:

$$(e_0, e_1) \in G_i \leftrightarrow e_0 \rightarrow_i e_1.$$

This is the standard way of representing relations (such as $\rightarrow_i$) in set theory. G_i can also be viewed as the set of edges of a directed graph (given by the transitive reduction of the partial order $\rightarrow_i^*$): this graph-theoretical view will sometimes be useful and gives us the relevant jargon, allowing us to refer to nodes instead of events, etc. We also denote with

$$G := \cup_i G_i$$

the union of all the immediate causality relations and with

$$Ev_i := \operatorname{dom} G_i \cup \operatorname{ran} G_i$$

the set of all the events of the i-th event structure (dom and ran take the domain and the range set, respectively, of any given relation). Finally,

$$Ev := \bigcup_i Ev_i$$

is the set of all the events.

3.1 Trace Selection

Recall that a configuration X_i for Ev_i is a set of events of Ev_i which is downward closed and conflict-free:

$$\forall e_0, e_1 \in Ev_i.\ e_0 \in X_i \wedge e_1 \in X_i \rightarrow \neg(e_0 \# e_1) \tag{1}$$

$$\forall e_1 \in \operatorname{ran}(G_i).\, e_1 \in X_i \rightarrow \bigwedge_{e_0 \in \left(G_i^{-1}\right)^{\rightarrow}\{e_1\}} e_0 \in X_i \tag{2}$$

Here $R^{\rightarrow}$ represents the function mapping a set to its image set through a given relation R. We note that, since G_i represents a relation as a set of ordered pairs, G_i^{-1} is the inverse relation. Further, recall that a configuration X_i is a trace exactly when it is maximal, that is,

$$\forall Y \subseteq Ev_i \text{ satisfying } (1)\text{–}(2)\ , Y \subseteq X_i. \tag{3}$$

Now, the quantification appearing in (3) is not directly expressible, because a quantification over relations is second-order logic, and any SMT-LIB theory is within first-order logic. The mathematical language used in (1), (2) and 3 allows us to convey ideas to the reader in a more compact and, hopefully, more understandable way than crude SMT-LIB code would. However, it also hides difficulties such the one just explained, arising from the gap between the expressiveness of first-order logic and that of standard mathematical notation. This problem can have several solutions: in [6] we used bit-vectors to represent sets and, in [7] we showed condition (3) to be equivalent to the following, first-order ones:

$$\forall e_1 \in Ev_i \backslash X_i.\ \exists e_0 \in Ev_i.\quad ((e_0 \# e_1 \wedge e_0 \in X_i) \vee ((e_0, e_1) \in G_i \wedge e_0 \notin X_i)) \tag{4}$$

This allows us to formulate the problem of trace finding in SMT-LIB.

3.2 Concurrent Execution of Traces

The next step is to define a measure of concurrency which will allow us to execute a given trace (or configuration) without violating the constraints given by the causality and conflict relations. Since any configuration is conflict-free, we only need to focus on not violating the causality relation when executing it.

1. Consider that every event in the configuration has a start time and an end time, with the latter not smaller than the former;
2. Any two ordered events should not overlap;
3. Any successor (according to $\rightarrow^*$) of an event should happen after it.

These conditions normally leave quite some freedom in the arrangement of the events for an execution as shown in Fig. 2 for a configuration X_1. A conceptually simple approach (commonly referred to as interleaving or sequentialising) is to find a linear order respecting $\rightarrow^*$: this means finding an order morphism from the configuration (seen as equipped with the partial order $\rightarrow^*$) to $\mathbb{Z}$ (equipped with its standard linear, total order). This way of handling execution is particularly well-suited to an SMT solver due to its awareness of integer arithmetics, and is the one we adopted in [7].

This approach is usually contrasted, in the literature, with the true concurrent one, whereby the restriction given by this linearisation is dropped, with an additional number of possible execution arrangements becoming possible which still comply with the three conditions above. This amounts to permitting to take advantage of possible parallelism, which gets lost in the interleaving process: in the latter approach, compliance with conditions (1)-(3) is attained by ruling out overlaps between any pair of distinct events, thereby including concurrent events (recall that two events are concurrent if they are not related by $\rightarrow^*$ nor by $\#$).

We now introduce an SMT way of

- modelling this increased choice of possible executions for the traces selected in the previous subsection and
- ranking them according to their efficiency, that is, their degree of concurrency.

It will rely on two functions, s and t, defined on all the events, and returning the time each of them starts and ends, respectively.

The first constraint we impose on them is that any single event must respect time:

$$\forall e \in Ev_i . t\left(e\right) - s\left(e\right) = \nu_2\left(e\right). \tag{5}$$

Let us recall that we assume that our event structures are finite so that, in particular, we can define a function p_i over the set of non-source nodes for the directed graph $\operatorname{ran} G_i$ returning, for each such node, its parent node in the considered trace X_i which terminates last:

$$\forall e_1 \in \operatorname{ran} G_i . \qquad \begin{cases} p_i\left(e_1\right) \in X_i \cap {G_i^{-1}}^{\rightarrow}\left(\{e_1\}\right) \\ \forall e_0 \in X_i \cap {G_i^{-1}}^{\rightarrow}\left(\{e_1\}\right) . \; t\left(p_i\left(e_1\right)\right) \geq t\left(e_0\right). \end{cases} \tag{6}$$

We have only one requirement to impose on p:

$$\forall e \in \operatorname{ran} G_i.\ s(e) \geq t(p(e)). \tag{7}$$

This last requirement can be tightened to $\forall e \in \operatorname{ran} G_i.\ s(e) = t(p(e))$ in case we are seeking for executions without idle times between subsequent events.

The formulas introduced in this section capture the linearly interleaved (or sequentialised) executions we described above, but add more possible executions. Intuitively, the former are the least possible concurrent execution, and in this sense they are also the least efficient ones. To make this kind of comparison more precise, we introduce a measure to rank all the different possible executions. This will give a metric, or distance, between executions according to their degree of concurrency. Consider, for a given G_i, the event which terminates last in a given execution of the trace X_i, denoted z_i; it can be described by the formulas

$$z_i \in X_i \cap (\operatorname{ran} G_i \setminus \operatorname{dom} G_i) \tag{8}$$

$$\forall e \in X_i \cap (\operatorname{ran} G_i \setminus \operatorname{dom} G_i).\ t(z_i) \geq t(e). \tag{9}$$

We will consider an execution of X_i less concurrent than another whenever $t(z_i)$ is larger for the former than for the latter. In this sense, we can use, e.g., an optimising SMT solver [17] to find the most concurrent execution by simply asking it to minimise the quantity $t(z_i)$. In particular, this definition of amount of concurrency can be reconciled with the intuition that sequentialised executions are the least concurrent ones: the number above will not be maximum for such executions, given a fixed X_i. This fact is also practically useful: since we know that a linearised execution has maximal duration, and since such duration is obviously bounded by the sum of all the event durations, we can pass this bound as an assertion to the SMT solver to reduce its search space when computing a generic execution.

Finally, among the several possible traces captured by the assertions of Section 3.1, we want to select the best one (see the example in Section 2). To do this, we consider one generic trace X_i for each G_i, and describe which labels are active at a given time y, via the function l:

$$l(y) = \bigcup_{e \in \bigcup_i X_i} b(\rho(e))\,(y \in [s(e), t(e)[).$$

Here, $\rho(e)$ describes the selected label for the event e (which is imposed to be in the set of allowed labels through an assertion we omit here), and $b(j)$ sets the j-th bit in a bit-vector of length equal to the overall number of all labels. Therefore, $l(y)$ is a bit-vector whose 1-bits correspond exactly to the labels active at time y, because the term in the rightmost pair of brackets filters exactly the active events (note that our convention is that the when the ending time $t(e)$ of an event ticks, we consider the event instantly off, so that the range in that term is right-open). We finally ask the SMT solver to maximise

$$\sum_{t=0}^{\max_i\{t(z_i)\}} \Gamma(l(t)),$$

where Γ is a function returning the interaction between the labels denoted by $l(t)$.

4 Verification

We have seen in Section 3.1 how the gap in expressiveness between standard mathematical notation and the first-order language of SMT solvers can cause difficulties when translating a mathematical problem for its computation in an SMT setting. In some cases, reformulations are needed (recall how we passed from (1) and (2) to (4) earlier). Even when the translation is more straightforward, several choices affect the final SMT code passed to the solver. One common choice is to instantiate universal quantifiers appearing in formulas exploiting the fact that they quantify over finite domains which are easy to compute separately, because this can considerably help the solver. For example, (9) features a universal quantifier ranging over a subset of the sinks of G_i, which is easily computed, for non-trivial graphs, as the set-theoretical difference between the set all children of some node of G_i and the set of all parents of some node of G_i. Performing this computation before invoking the SMT solver helps by removing a universal quantifier and by carefully restricting the possible cases over which it is instantiated; this kind of pre-processing is done often to obtain better performing SMT problems, but comes with a significant drawback: the SMT code thus obtained becomes exceedingly verbose, and, typically, difficult to peruse. In other words, the final SMT code usually looks extremely different (and extremely longer) than the original pen-and-paper mathematical formulation of the problem; in our case, we can compare the concise mathematical formulation in Section 3 with the final SMT code we effectively run for a simple example, a small (necessarily truncated) excerpt of which is in Listing 1.1, representing (6) and (7).

Listing 1.1. SMT macro for (6) and (7), in the case of a small example (excerpt)

```
(define-fun trueConcurrencySMT () Bool
 (let (($x206 (and true (=> (isSelected g0)
 (>= (endTime (lastParent g1)) (endTime g0))))))
 (let (($x212 (and (or false (and (isSelected g0)
 (= (lastParent g1) g0))) $x206 (= (idleTime g1)
 (- (startTime g1) (endTime (lastParent g1))))
 (>= (idleTime g1) 0)))) (let (($x165 (isSelected g1)))
 (let (($x195 (and (or false (and $x165
 (= (lastParent g2) g1)))(and true (=> $x165
 (>= (endTime (lastParent g2)) (endTime g1))))
 (= (idleTime g2) (- (startTime g2) (endTime (lastParent g2))))
 (>= (idleTime g2) 0)))) (let (($x142 (isSelected g2)))
 (let (($x180 (and (or false (and $x165 (= (lastParent g3) g1)))
 (and true (=> $x165 (>= (endTime (lastParent g3)) (endTime g1))))
 (= (idleTime g3) (- (startTime g3) (endTime (lastParent g3))))
 (>= (idleTime g3) 0))))
 (let (($x146 (isSelected g3))) [...]
```

The striking difference between them is clearly a problem when examining the SMT code and in particular when this code needs to have some degree of confidence in the result it produces.

However, nobody prevents us from keeping two SMT codes: one (let us call it code A) closer to the mathematical formulation (and likely less efficient), and the other (code B) which underwent a series of transformations as hinted just above. If the SMT solver returns no models satisfying one code and not the other, then we can be confident about their equivalence, assuming correctness of the solver. But, more than that, code A is usually compact enough to be easily formulated within a theorem prover whose foundations entail the first-order logic (plus arithmetics) featured in an SMT solver, such as Isabelle/HOL [20]. And, if we manage to do that, we can use existing SMT code generators from within the theorem prover, which means that the definitions formulated in the theorem provers can be used both to prove their correctness and to generate code A. On top of that, we can arbitrarily introduce intermediate equivalent codes between A and B, if this eases the verification process. We now see a particular application of this general verification mechanism.

In Isabelle/HOL, we can introduce the following function:

```
abbreviation "trueConcur01' G X lastParent startTime endTime
≡ ∀ child. (X child & (∃ parent. G parent child)) →
(X (lastParent child) & (G (lastParent child) child) &
(∀ parent. (X parent & G parent child) →
        endTime(lastParent child) ≥ endTime parent) &
startTime(child) ≥ endTime(lastParent(child)))"
```

It is carefully crafted to be equivalent, as the one in listing 1.1, to (6) and (7) and to only feature mathematical objects available in the first-order logic of an SMT solver; this means no sets, no higher-order functions, no lists, etc: everything is represented as functions and predicates (for example, instead of writing `child` ∈ `X`, we write `X child`, where `X` is a predicate, or boolean function). This allows us, on one hand, to export it to an SMT constant (let us call it trueConcurrencyIsabelle) taking advantage of the translator provided by Isabelle's *Sledgehammer* tool [4], and to check its equivalence to trueConcurrencySMT inside an SMT solver (expecting the answer `unsat`):

```
(assert (or
  (and trueConcurrencySMT (not trueConcurrencyIsabelle))
  (and (not trueConcurrencySMT) (trueConcurrencyIsabelle))
  )
)
```

On the other hand, the Isabelle definition can be proved correctness theorems about, thereby granting automatically that such correctness proofs carry over to trueConcurrencySMT. Let us focus on a basic property we expect from any execution, namely that expressed by requirements (2) and (3) of Section 3.2. The first step is usually to formulate the SMT-friendly Isabelle definition above into an equivalent one, but possibly more convenient when coming to formal proofs. This does not compromise the verification efforts because these new definitions

can be provided with formal theorems proving their equivalence to the original ones. In this case, we will use

```
abbreviation
"TrueConcur00' P X lastParent startTime endTime ==
∀ child ∈ X ∩ (Range P). (lastParent(child) ∈ X ∩
immediatePredecessors' P {child} &
endTime(lastParent(child)) ≥
  Max(endTime`(X ∩ (immediatePredecessors' P {child}))) &
startTime(child) ge endTime(lastParent(child)))",
```

all the relations have been expressed as sets of ordered pairs, the immediate causality relation `G` (corresponding to $\rightarrow$) has been substituted by its transitive-reflexive closure `P` (corresponding to $\rightarrow^*$), and the infix operator ` takes the image of a set through a function. We note that the custom-defined operator `immediatePredecessors` only makes sense for discrete relations; this is not a problem here, because we are in a finite setting. We omit the relevant equivalence proof, and only state the main correctness formal theorem:

```
theorem fixes startTime:: "_ => nat" assumes "wf (strict P)"
"wf (strict (P^-1))" "atomicTimeArrow P startTime endTime"
"isPo P"
"TrueConcur00' P X lastParent startTime endTime" shows
"{e2 ∈ X ∩ events P. e2 |
  (∃ e0 ∈ X ∩ (strict P)^-1``{e2}.
   startTime(e2)<endTime(e0))}={}"
```

The thesis (introduced by the `shows` keyword) states that the set of the events in `X` possibly admitting a predecessor ending after they start is empty (please note that `(strict P)^-1` is the strict causality relation reversed, and ``` ``{e2} ``` takes all the images of the singleton `{e2}` through it, thereby meaning all the predecessors of `e2` through `P`). Between the keywords `assumes` and `shows` there are five hypotheses. `atomicTimeArrow` formalises requirement (5), while `isPo`' ensures that `P` is a partial order. Finally, `wf` is a predicate returning whether or not a given relation is *well-founded*, meaning that it supports transfinite induction [11]. Since this property is trivially entailed by finiteness, we showed that our correctness result holds in a more general setting than the one we employ it on.

To keep this generality, we had to prove several additional formal theorems about well-foundedness, some of independent interest. The following one, for example, is fundamental to the previous one, and states the intuitive idea that between two elements related by a well-founded order relation `P`, there must be a third one which is immediately related to one of them:

```
lemma mm05ll: assumes "reflex P" "trans P"
"antisym P" "wf (strict P)" "(x,z) in P"
"z ≠ x" shows
"∃ y. ((y,z) ∈ P & y ∈ next1 P {x})".
```

Here, `next1 P {x}` denotes the immediate successors of x according to the order relation `P`: the lemma states that for well-ordered relations, this function yields what is expected.

To recapitulate, our general approach to verifying SMT-LIB code proposes a chain of equivalent definitions, with one end expressed in Isabelle/HOL and the other directly in SMT-LIB. Each of them is proven equivalent to the next either using the SMT solver or a formal Isabelle proof. At one point in the chain, we pass from an Isabelle definition to an SMT-LIB one: here, we trust Sledgehammer to produce a first-order SMT-LIB definition from an Isabelle/HOL definition restricted to the first-order fragment of HOL.

This very last passage raises a key question: why should we trust Sledgehammer in building the formal proofs of our SMT-LIB code? After all, Isabelle itself does not trust Sledgehammer, but only uses its results to obtain guidance from various solvers, with the final proof thus obtained re-checked for correctness inside Isabelle.

The answer starts from a basic observation: ultimately, verification is about reducing the amount of code to be trusted. Ideally, to a core small enough to be uncontroversial (see the de Bruijn criterion [2] which, nevertheless, implies trusting software/hardware below small core). But even reducing the trusted code from the extremely verbose SMT-LIB code to a limited amount of Sledgehammer's ML code is desirable, for various reasons. First, the latter stays the same when different SMT-LIB problems are considered. Second, possible soundness problems are linked to the fact that higher-order logic is much richer than SMT-LIB first-order logic: in our approach, this issue is greatly mitigated by the fact that we use Sledgehammer translator for code (e.g., trueConcur01') which only uses a first-order logic fragment of HOL, while the type-richer definitions are confined to duplicate Isabelle definitions (e.g., trueConcur00') which are used for theorem proving and which are proved equivalent, *within* Isabelle, to the first-order definitions seen by the translator generating SMT code. Third, the limited portion of the SMT translator we need has had much more exposure and testing than any particular SMT-LIB code that we would need to trust without this approach. Fourth, ML (the functional, higher order language in which Isabelle and Sledgehammer are implemented) is arguably more readable than SMT-LIB. Fifth, contrary to the Isabelle case, we need to trust the SMT solver anyway because we are not using it to obtain proofs (which Isabelle can then reconstruct), but computations. And the size and complexity of the Sledgehammer implementation is vastly negligible in comparison.

5 Conclusions

We presented an SMT formulation for finding optimal true concurrent traces of execution. The typical issue of the SMT code getting rapidly unreadable is addressed by using a theorem prover to generate equivalent SMT assertions and then to prove correctness theorem about the latter. In previous work, we explored the problem of searching optimal traces of executions subject to linear interleaving. Here, we provided a novel approach to use an SMT solver for finding executions of optimal traces with a varying degree of concurrency. Future work could see

the application of a similar approach to resource scheduling and optimisation problems.

This paper also contributes a way of characterising and computing a notion of degree of parallelism for arbitrary computations. Our definition has some advantages over existing ones: contrary to [22], it does not require an empirical implementation on given hardware in order to analyse and compute the degree of parallelism of a given algorithm; it does not require to express the algorithm in a specific language, as in [21]; it reduces to computing a numerical value through a simple procedure, rather than introducing definitions (see functionals in Sections 1 and 2 of [21]) which are very useful for abstract analysis but, to the best of our knowledge, currently have no concrete implementation.

To make our work more useful, we plan to introduce additional metrics: besides the degree of concurrency, an user could find useful to have further ways of comparing possible executions. For example, by considering how many events two possible solutions have in common, and what is their time distance.

We also plan to investigate possible connections between the SMT formalisation proposed here and other framework formalising concepts related to the ones emerging in this paper; for example, in Event Calculus [13], events and time are central.

Finally, we will continue exploiting the interplay between SMT solvers and theorem provers. As shown in the current and previous works, this interaction can be quite general and fertile. However, it has to be adapted on a case by case basis. This could be made easier by providing systematic, generic procedures to automate it at least in its most common use cases, for example the instantiation of a universal quantifier, which we faced in Section 4.

References

1. Paolo Baldan and Silvia Crafa. A logic for true concurrency. In *International Conference on Concurrency Theory*, pages 147–161. Springer, 2010.
2. Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. *Handbook of automated reasoning*, 2:1149–1238, 2001.
3. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.
4. Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C Paulson. Extending Sledgehammer with SMT solvers. *Journal of automated reasoning*, 51(1):109–128, 2013.
5. J. K. F. Bowles and M. B. Caminati. Balancing prescriptions with constraint solvers. In P. Liò and P. Zuliani, editors, *Automated Reasoning for Systems Biology and Medicine*, volume 30 of *Computational Biology*, pages 243–267. Springer, 2019.
6. J. K. F. Bowles and M. B. Caminati. An integrated approach to a combinatorial optimisation problem. In *Integrated Formal Methods (iFM 2019)*, volume 11918 of *LNCS*, pages 284–302, 2019.
7. J. K. F. Bowles and M. B. Caminati. Correct composition in the presence of behavioural conflicts and dephasing. *Science of Computer Programming*, 185, 2020.

8. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*, volume 4350. Springer, 2007.
9. Francisco Durán, Camilo Rocha, and Gwen Salaün. Computing the parallelism degree of timed bpmn processes. In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 320–335. Springer, 2018.
10. Vashti C Galpin. Equivalence semantics for concurrency: comparison and application. 1998.
11. Egbert Harzheim. *Ordered sets*, volume 7. Springer Science & Business Media, 2006.
12. Joost-Pieter Katoen, Christel Baier, and Diego Latella. Metric semantics for true concurrent real time. *Theoretical Computer Science*, 254(1-2):501–542, 2001.
13. Robert Kowalski and Marek Sergot. A logic-based calculus of events. In *Foundations of knowledge base management*, pages 23–55. Springer, 1989.
14. J. Küster-Filipe. Modelling concurrent interactions. *Theoretical Computer Science*, 351:203–220, 2006.
15. Kamal Lodaya, Madhavan Mukund, Ramaswamy Ramanujam, and PS Thiagarajan. Models and logics for true concurrency. *Sadhana*, 17(1):131–165, 1992.
16. Philipp J Meyer, Javier Esparza, and Hagen Völzer. Computing the concurrency threshold of sound free-choice workflow nets. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–19. Springer, 2018.
17. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
18. M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part i. *TCS*, 13:85–108, 1981.
19. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
20. Tobias Nipkow, L. C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, London, UK, 2002.
21. V Yu Sazonov. Degrees of parallelism in computations. In *International Symposium on Mathematical Foundations of Computer Science*, pages 517–523. Springer, 1976.
22. K Sevcik. Characterizations of parallelism in applications and their use in scheduling. *Perform. Eval.*, 17:171–180, 1989.
23. G. Winskel. Event structure semantics for CCS and related languages. In M. Nielsen and E.M. Schmidt, editors, *Automata, Languages, and Programming*, volume 140 of *LNCS*, pages 561–576. Springer, 1982.
24. G. Winskel and M. Nielsen. Models for Concurrency. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4, Semantic Modelling*, pages 1–148. Oxford Science Publications, 1995.